# CSE 311 Lecture 28: Undecidability of the Halting Problem

Emina Torlak and Kevin Zatloukal

# Topics

**Final exam**

Logistics, format, and topics.

**Countability and uncomputability**

A quick recap of Lecture 27.

**Undecidability of the halting problem**

Important problems computers can’t solve.

# Final exam

Logistics, format, and topics.

# Final exam logistics

**Monday, December 10 in JHN 102**

Section A at 16:30-18:20,

Section B at 14:30-16:20.

# Final exam logistics

**Monday, December 10 in JHN 102**

Section A at 16:30-18:20,

Section B at 14:30-16:20.

**Important: take the exam at the time assigned to your section.**

If you have a scheduling conflict, email the staff ASAP.

Bring your UW ID and have it ready to be checked during the exam.

# Final exam logistics

**Monday, December 10 in JHN 102**

Section A at 16:30-18:20,

Section B at 14:30-16:20.

**Important: take the exam at the time assigned to your section.**

If you have a scheduling conflict, email the staff ASAP.

Bring your UW ID and have it ready to be checked during the exam.

**Final review session is on Sunday, Dec 09 at 15:00-17:00 in GWN 301.**

Bring your questions!

# Final exam format and topics

## Format

**8 problems** in **110 minutes**.

Closed book, closed notes, no calculators, no cellphones.

# Final exam format and topics

**Format**

**8 problems** in **110 minutes**.
Closed book, closed notes, no calculators, no cellphones.

**Questions and topics**

(1) Write a regex, DFA, CFG for given languages.
(2) A formal proof.
(3) A proof by strong induction.
(4) A proof by structural induction.
(5) NFA to DFA conversion.
(6) DFA minimization.
(7) A proof that a language is irregular.
(8) Short answers on modular arithmetic, relations, logic, uncomputability.

# Final exam format and topics

## Format

8 problems in 110 minutes.

Closed book, closed notes, no calculators, no cellphones.

## Questions and topics

(1) Write a regex, DFA, CFG for given languages.
(2) A formal proof.
(3) A proof by strong induction.
(4) A proof by structural induction.
(5) NFA to DFA conversion.
(6) DFA minimization.
(7) A proof that a language is irregular.
(8) Short answers on modular arithmetic, relations, logic, uncomputability.

You've solved similar problems on homeworks and in sections.

**Do the easy parts of all the problems first.** Don't get stuck on one problem!

# Countability and uncomputability

A quick recap of Lecture 27.

# Countable and uncountable sets

**Countable set**

A set is *countable* iff it has the same cardinality as some subset of $\mathbb{N}$.

# Countable and uncountable sets

**Countable set**

A set is *countable* iff it has the same cardinality as some subset of $\mathbb{N}$.

**Countable sets**

- $\mathbb{N}$ (natural numbers)
- $\mathbb{Z}$ (integers)
- $\mathbb{Q}^+$ (positive rational numbers)
- $\Sigma^*$ over finite $\Sigma$
- All (Java) programs

Shown by dovetailing.

# Countable and uncountable sets

**Countable set**

A set is *countable* iff it has the same cardinality as some subset of $\mathbb{N}$.

**Countable sets**

- $\mathbb{N}$ (natural numbers)
- $\mathbb{Z}$ (integers)
- $\mathbb{Q}^+$ (positive rational numbers)
- $\Sigma^*$ over finite $\Sigma$
- All (Java) programs

Shown by dovetailing.

**Uncountable sets**

- All real numbers in $[0, 1)$
- All functions from $\mathbb{N}$ to $\{0, 1\}$

Shown by diagonalization.

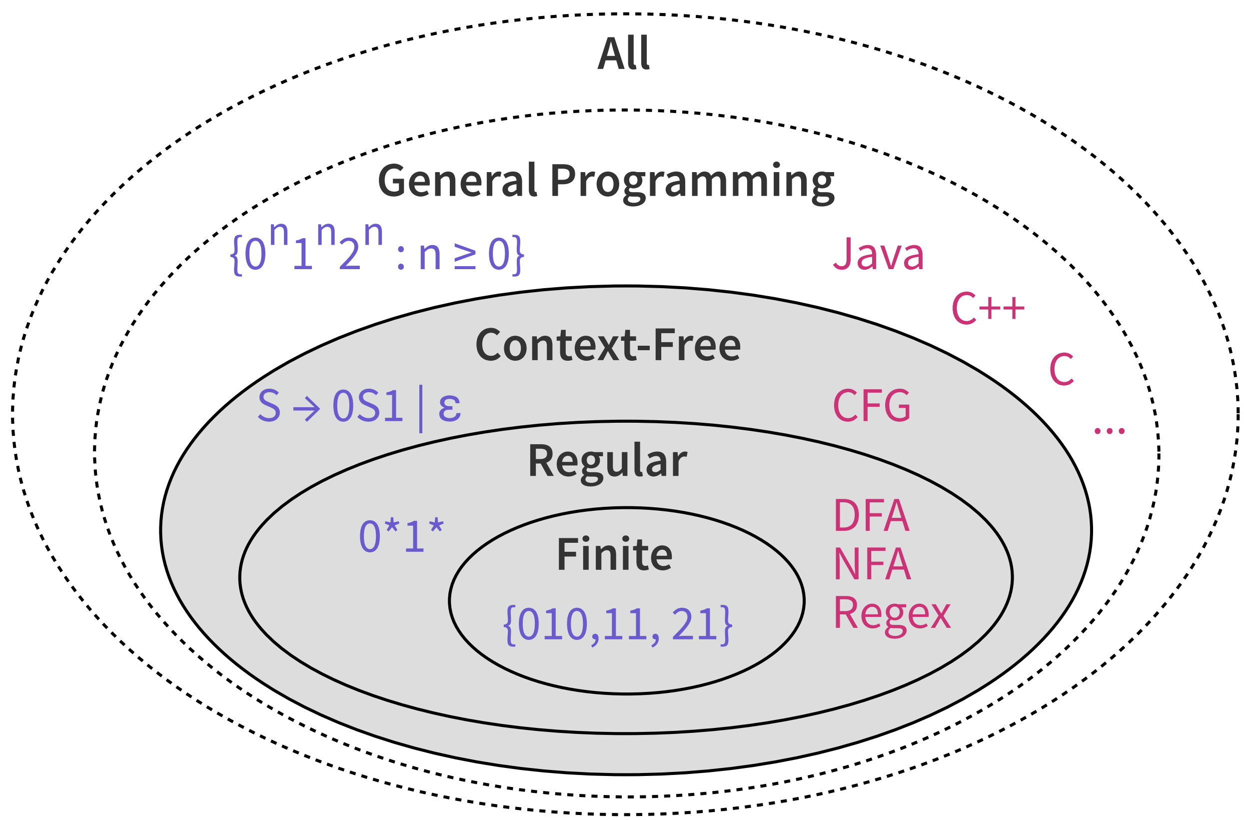
# Uncomputable functions

**We have seen that ...**

The set of all (Java) programs is countable.

The set of all functions $f : \mathbb{N} \rightarrow \{0, 1\}$ is uncountable.

So there must be some function that is not computable by any program!



# Uncomputable functions

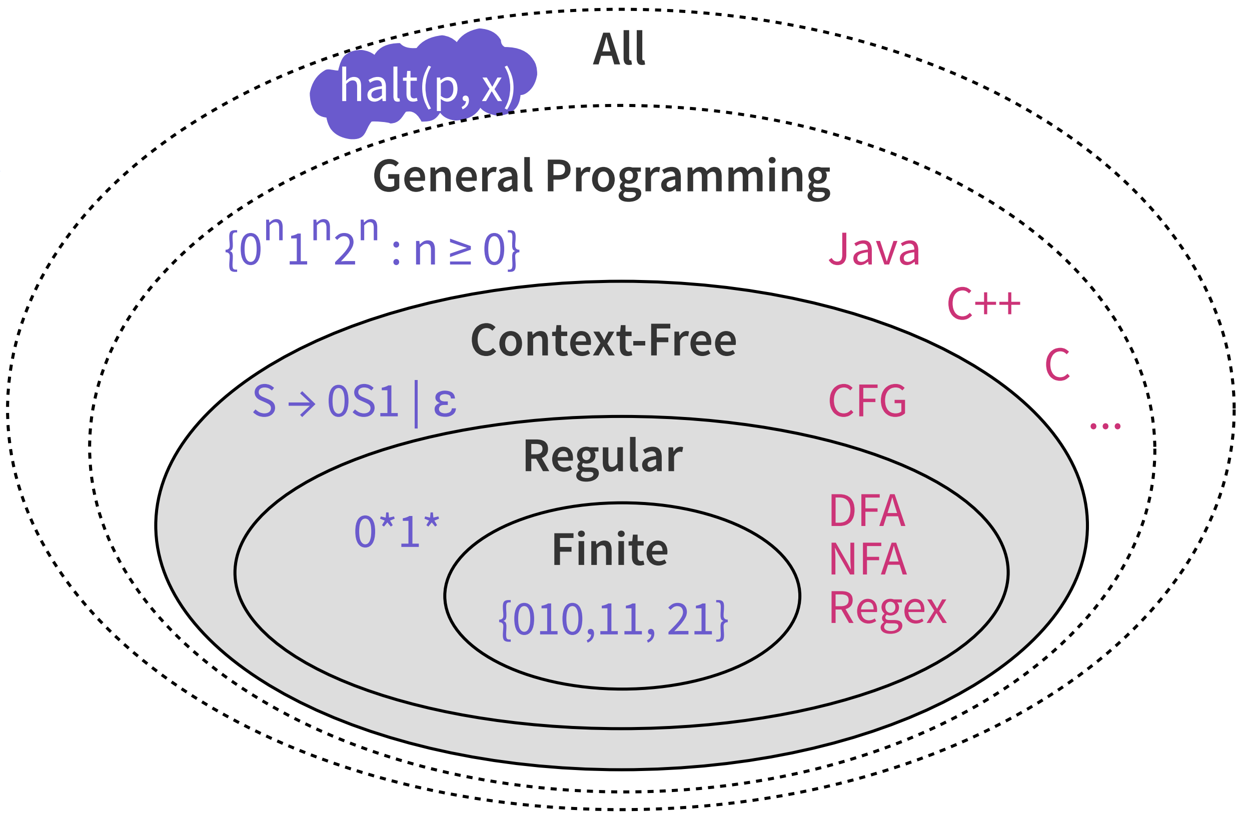
**We have seen that ...**

The set of all (Java) programs is countable.

The set of all functions $f : \mathbb{N} \rightarrow \{0, 1\}$ is uncountable.

So there must be some function that is not computable by any program!

We'll study one such important function today.



# Undecidability of the halting problem

Important problems computers can’t solve.

# First, some notation …

**We'll be talking about (Java) code.**

`code(P)` will denote "the code of the program `P`".

# First, some notation …

**We'll be talking about (Java) code.**

`code(P)` will denote "the code of the program `P`".

**Consider this program:**

```
public static boolean P(String x) {
  return x.matches("public .*");
}
```

# First, some notation …

**We'll be talking about (Java) code.**

`code(P)` will denote "the code of the program `P`".

**Consider this program:**

```
public static boolean P(String x) {
  return x.matches("public .*");
}
```

**What is `P(code(P))`?**

# First, some notation …

**We'll be talking about (Java) code.**

`code(P)` will denote "the code of the program `P`".

**Consider this program:**

```
public static boolean P(String x) {
  return x.matches("public .*");
}
```

**What is `P(code(P))`?**

true

# And now, the halting problem!

**The Halting Problem**

Given an input $x$ and `code(P)` for any program $P$,
output `true` if $P$ halts on the input $x$, and
output `false` if $P$ does not halt (diverges) on the input $x$.

# And now, the halting problem!

**The Halting Problem**

Given an input $x$ and `code(P)` for any program $P$,
output `true` if $P$ halts on the input $x$, and
output `false` if $P$ does not halt (diverges) on the input $x$.

**Can't we determine this by just running $\mathbf{P}$ on $\mathbf{x}$?**

# And now, the halting problem!

**The Halting Problem**

Given an input $x$ and `code(P)` for any program $P$,
output `true` if $P$ halts on the input $x$, and
output `false` if $P$ does not halt (diverges) on the input $x$.

**Can't we determine this by just running $\mathbf{P}$ on $\mathbf{x}$?**

No! We can't tell if $P$ diverged on $x$ or is taking a long time to return.

# The halting problem is undecidable

**The Halting Problem**

Given an input $x$ and `code(P)` for any program $P$,
output `true` if $P$ halts on the input $x$, and
output `false` if $P$ does not halt (diverges) on the input $x$.

**Theorem (due to Alan Turing)**

There is no program that solves the halting problem.

# The halting problem is undecidable

**The Halting Problem**

Given an input $x$ and `code(P)` for any program $P$,
output `true` if $P$ halts on the input $x$, and
output `false` if $P$ does not halt (diverges) on the input $x$.

**Theorem (due to Alan Turing)**

There is no program that solves the halting problem.

In other words, there is no program that computes the function described by the halting problem. This function is therefore uncomputable. Because the function outputs a boolean (a yes/no decision), we say that the underlying problem is *undecidable*.

# Proof by contradiction

**Suppose that `H` is a program that solves the halting problem.**

# Proof by contradiction

**Suppose that `H` is a program that solves the halting problem.**

Then, we can write the program `D` as follows:

```
public static void D(String x) {
  if (H(x, x) == true) {
    while (true); // diverge
  } else {
    return;       // halt
  }
}
```

# Proof by contradiction

**Suppose that `H` is a program that solves the halting problem.**

Then, we can write the program `D` as follows:

```
public static void D(String x) {
  if (H(x, x) == true) {
    while (true); // diverge
  } else {
    return;       // halt
  }
}
```

Does `D(code(D))` halt?

# Proof by contradiction

**Suppose that `H` is a program that solves the halting problem.**

Then, we can write the program `D` as follows:

```
public static void D(String x) {
  if (H(x, x) == true) {
    while (true); // diverge
  } else {
    return;       // halt
  }
}
```

Does `D(code(D))` halt?

**`H` solves the halting problem means the following:**

If `D(x)` halts then `H(code(D),x)` is true otherwise `H(code(D),x)` is false.

# Proof by contradiction

**Suppose that `H` is a program that solves the halting problem.**

Then, we can write the program `D` as follows:

```
public static void D(String x) {
  if (H(x, x) == true) {
    while (true); // diverge
  } else {
    return;       // halt
  }
}
```

Does `D(code(D))` halt?

**`H` solves the halting problem means the following:**

If `D(x)` halts then `H(code(D),x)` is true otherwise `H(code(D),x)` is false.

**Suppose that `D(code(D))` halts.**

Then, by definition of `H`, it must be that `H(code(D),code(D))` is true.

But in that case, `D(code(D))` doesn't halt by definition of `D`.

# Proof by contradiction

**Suppose that `H` is a program that solves the halting problem.**

Then, we can write the program `D` as follows:

```
public static void D(String x) {
  if (H(x, x) == true) {
    while (true); // diverge
  } else {
    return;       // halt
  }
}
```

Does `D(code(D))` halt?

**`H` solves the halting problem means the following:**

If `D(x)` halts then `H(code(D),x)` is true otherwise `H(code(D),x)` is false.

**Suppose that `D(code(D))` halts.**

Then, by definition of `H`, it must be that `H(code(D),code(D))` is true.

But in that case, `D(code(D))` doesn't halt by definition of `D`.

**Suppose that `D(code(D))` doesn't halt.**

Then, by definition of `H`, it must be that `H(code(D),code(D))` is false.

But in that case, `D(code(D))` halts by definition of `D`.

# Proof by contradiction

**Suppose that `H` is a program that solves the halting problem.**

Then, we can write the program `D` as follows:

```
public static void D(String x) {
  if (H(x, x) == true) {
    while (true); // diverge
  } else {
    return;       // halt
  }
}
```

Does `D(code(D))` halt?

**`H` solves the halting problem means the following:**

If `D(x)` halts then `H(code(D),x)` is true otherwise `H(code(D),x)` is false.

**Suppose that `D(code(D))` halts.**

Then, by definition of `H`, it must be that `H(code(D),code(D))` is true.

But in that case, `D(code(D))` doesn't halt by definition of `D`.

**Suppose that `D(code(D))` doesn't halt.**

Then, by definition of `H`, it must be that `H(code(D),code(D))` is false.

But in that case, `D(code(D))` halts by definition of `D`.

**So we reach a contradiction in either case.**

Therefore, our assumption that `H` exists must be false. □

# Where did the idea for creating `D` come from?

```
public static void D(Object x) {
  if (H(x, x) == true) {
    while (true); // diverge
  } else {
    return;       // halt
  }
}
```

**Note that `D` halts on `code(P)`**

iff `H(code(P),code(P))` outputs false, i.e.,

iff `P` doesn't halt on the input `code(P)`.

**Therefore, `D` differs from every program `P` on the input `code(P)`.**

# Where did the idea for creating `D` come from?

```
public static void D(Object x) {
  if (H(x, x) == true) {
    while (true); // diverge
  } else {
    return;       // halt
  }
}
```

**Note that `D` halts on `code(P)`**

iff `H(code(P),code(P))` outputs false, i.e.,

iff `P` doesn't halt on the input `code(P)`.

**Therefore, `D` differs from every program `P` on the input `code(P)`.**

This sounds like diagonalization!

# “D” is for diagonalization

**List all Java programs.**

This list exists because the set of all Java programs is countable.

| | |
|---|---|
| $P_0$ | |
| $P_1$ | |
| $P_2$ | |
| $P_3$ | |
| $P_4$ | |

# "D" is for diagonalization

**List all Java programs.**

This list exists because the set of all Java programs is countable.

**Let $\langle \mathbf{P} \rangle$ stand for `code(P)`.**

| | $\langle P_0 \rangle$ | $\langle P_1 \rangle$ | $\langle P_2 \rangle$ | $\langle P_3 \rangle$ | $\langle P_4 \rangle$ | $\ldots$ |
|---|---|---|---|---|---|---|
| $P_0$ | | | | | | |
| $P_1$ | | | | | | |
| $P_2$ | | | | | | |
| $P_3$ | | | | | | |
| $P_4$ | | | | | | |

# “D” is for diagonalization

**List all Java programs.**

This list exists because the set of all Java programs is countable.

**Let $\langle \mathbf{P} \rangle$ stand for `code(P)`.**

$(P, x)$ entry is 1 if the program $P$ halts on input $x$ and 0 otherwise.

| | $\langle P_0 \rangle$ | $\langle P_1 \rangle$ | $\langle P_2 \rangle$ | $\langle P_3 \rangle$ | $\langle P_4 \rangle$ | $\ldots$ |
|---|---|---|---|---|---|---|
| $P_0$ | 0 | 1 | 1 | 0 | 1 | $\ldots$ |
| $P_1$ | 1 | 1 | 0 | 1 | 0 | $\ldots$ |
| $P_2$ | 1 | 0 | 1 | 0 | 0 | $\ldots$ |
| $P_3$ | 0 | 1 | 1 | 0 | 1 | $\ldots$ |
| $P_4$ | 0 | 1 | 1 | 1 | 1 | $\ldots$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\ldots$ |

# "D" is for diagonalization

**List all Java programs.**

This list exists because the set of all Java programs is countable.

**Let $\langle \mathbf{P} \rangle$ stand for `code(P)`.**

$(P, x)$ entry is 1 if the program $P$ halts on input $x$ and 0 otherwise.

**`D` behaves like the flipped diagonal**

$D(\langle P \rangle) = \neg P(\langle P \rangle)$, and differs from every $P$ in the list.

| | $\langle P_0 \rangle$ | $\langle P_1 \rangle$ | $\langle P_2 \rangle$ | $\langle P_3 \rangle$ | $\langle P_4 \rangle$ | $\dots$ |
|---|---|---|---|---|---|---|
| $P_0$ | 0 | 1 | 1 | 0 | 1 | $\dots$ |
| $P_1$ | 1 | 1 | 0 | 1 | 0 | $\dots$ |
| $P_2$ | 1 | 0 | 1 | 0 | 0 | $\dots$ |
| $P_3$ | 0 | 1 | 1 | 0 | 1 | $\dots$ |
| $P_4$ | 0 | 1 | 1 | 1 | 1 | $\dots$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\dots$ |

# "D" is for diagonalization

**List all Java programs.**

This list exists because the set of all Java programs is countable.

**Let $\langle \mathbf{P} \rangle$ stand for `code(P)`.**

$(P, x)$ entry is 1 if the program $P$ halts on input $x$ and 0 otherwise.

**`D` behaves like the flipped diagonal**

$D(\langle P \rangle) = \neg P(\langle P \rangle)$, and differs from every $P$ in the list.

**But the list is complete.**

So if `D` isn't included, it cannot exist!

| | $\langle P_0 \rangle$ | $\langle P_1 \rangle$ | $\langle P_2 \rangle$ | $\langle P_3 \rangle$ | $\langle P_4 \rangle$ | $\ldots$ |
|---|---|---|---|---|---|---|
| $P_0$ | 0 | 1 | 1 | 0 | 1 | $\ldots$ |
| $P_1$ | 1 | 1 | 0 | 1 | 0 | $\ldots$ |
| $P_2$ | 1 | 0 | 1 | 0 | 0 | $\ldots$ |
| $P_3$ | 0 | 1 | 1 | 0 | 1 | $\ldots$ |
| $P_4$ | 0 | 1 | 1 | 1 | 1 | $\ldots$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\ldots$ |

# The halting problem isn't the only hard problem

**To show that a problem $\mathbf{B}$ is undecidable:**

Prove that if there were a program deciding $B$ then there would be a way to build a program deciding the halting problem.

# The halting problem isn't the only hard problem

**To show that a problem $\mathbf{B}$ is undecidable:**

Prove that if there were a program deciding $B$ then there would be a way to build a program deciding the halting problem.

That is, prove "$B$ is decidable $\rightarrow$ halting problem is decidable".

# The halting problem isn't the only hard problem

**To show that a problem $\mathbf{B}$ is undecidable:**

Prove that if there were a program deciding $B$ then there would be a way to build a program deciding the halting problem.

That is, prove "$B$ is decidable $\to$ halting problem is decidable".

By contrapositive, "halting problem is undecidable $\to$ $B$ is undecidable".

# The halting problem isn't the only hard problem

**To show that a problem $\mathbf{B}$ is undecidable:**

Prove that if there were a program deciding $B$ then there would be a way to build a program deciding the halting problem.
That is, prove "$B$ is decidable $\rightarrow$ halting problem is decidable".
By contrapositive, "halting problem is undecidable $\rightarrow$ $B$ is undecidable".
Therefore, $B$ is undecidable.

# The halting problem isn't the only hard problem

**To show that a problem $\mathbf{B}$ is undecidable:**

Prove that if there were a program deciding $B$ then there would be a way to build a program deciding the halting problem.
That is, prove "$B$ is decidable $\rightarrow$ halting problem is decidable".
By contrapositive, "halting problem is undecidable $\rightarrow$ $B$ is undecidable".
Therefore, $B$ is undecidable.

**Every non-trivial question about program behavior is undecidable.** ☹

Termination, equivalence checking, verification, synthesis, …

# The halting problem isn't the only hard problem

**To show that a problem $\mathbf{B}$ is undecidable:**

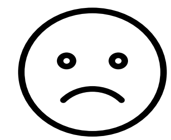
Prove that if there were a program deciding $B$ then there would be a way to build a program deciding the halting problem.

That is, prove "$B$ is decidable $\rightarrow$ halting problem is decidable".

By contrapositive, "halting problem is undecidable $\rightarrow$ $B$ is undecidable".

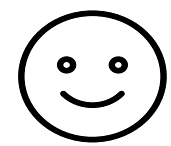
Therefore, $B$ is undecidable.

**Every non-trivial question about program behavior is undecidable.**



Termination, equivalence checking, verification, synthesis, …

**But we can often decide these questions in practice!**



They are undecidable for *arbitrary* programs and properties.

Yet decidable for many specific classes of programs and properties.

And when we allow "yes/no/don't know" answers.

# That's all folks!

Propositional logic.
Boolean logic, circuits, and algebra.
Predicates, quantifiers and predicate logic.
Inference rules and formal proofs for propositional and predicate logic.
English proofs.
Set theory.
Modular arithmetic and prime numbers.
GCD, Euclid's algorithm, modular inverse, and exponentiation.
Induction and strong induction.
Recursively defined functions and sets.
Structural induction.
Regular expressions.
Context-free grammars and languages.
Relations, composition, and reflexive-transitive closure.
DFAs, NFAs, and product construction for DFAs.
Finite state machines with output.
Minimization algorithm for finite state machines.
Conversion of regular expressions to NFAs.
Subset construction to convert NFAs to DFAs.
Equivalence of DFAs, NFAs, regular expressions.
Method to prove languages are not regular.
Cardinality, countability, and diagonalization.
Undecidability and the halting problem.

Go forth and prove great things!